Documentação do Trabalho Prático 3

Luísa Ribeiro Bezerra - Matrícula: 2015086441

Julho, 2017

1. Introdução

O objetivo principal deste trabalho é desenvolver nossa perspectiva e aprendizado sobre a diferenciação entre resolver um mesmo problema utilizando algoritmos baseados nos métodos Força Bruta, Guloso e Programação Dinâmica.

No Trabalho Prático sugerido, temos a realização de uma festa comemorativa da rua Pai no Bar, que possui vários bares. Os donos dos bares também residem nesta rua e, por serem pessoas muito desconfiadas, suas casas ficam sempre do lado da rua oposto ao bar (facilitando assim avistar movimentação estranha em sua casa ou bar). Com a Copa na Rússia chegando, os moradores da rua querem manter sua tradição de pendurar bandeirolas, porém não estão com verba para realizarem isso por causa da crise. Para tal, foi sugerido que fossem usadas as bandeirolas que os donos dos bares usam nos seus estabelecimentos - eles concordaram com a ideia, pois a rua enfeitada atrai movimento e consequentemente clientes. Eles concordaram em emprestar suas respectivas bandeirolas para enfeitar a rua com as condições a seguir:

• Cada bar vai contribuir com uma linha de bandeirola;

• As bandeirolas serão penduradas da seguinte maneira: uma ponta da linha no bar e a outra ponta na casa do dono do bar.

• As linhas não podem se cruzar.

Cada bar não necessariamente fica em frente a casa do seu dono - um bar pode estar na esquina, e a casa do dono na esquina oposta. A casa do dono do bar pode estar em qualquer lugar desde que seja do outro lado da rua onde está o bar. Em um lado da rua, temos apenas números ímpares, e do outro lado da rua, apenas números pares, e os números seguem em ordem crescente em ambos os lados da rua e na mesma direção. Dada a explicação, a proposta para realizar o trabalho seria de pendurar o maior número de linhas de bandeirola possível respeitando as condições dadas acima.

Durante a realização deste trabalho, foi estudado os três métodos indicados para resolver o problema, e o desenvolvimento de cada um deles está no próximo tópico (2.1, 2.2 e 2.3), mostrando detalhadamente todo o processo de criação da ideia em alto nível.

2. Desenvolvimento

O começo do código dá-se no *main.c*, onde temos a leitura das duas primeiras linhas da entrada dada - a letra (char), que pode ser g, b ou d, indicando método guloso, força bruta ou programação dinâmica. Após isso, lê-se um inteiro que representa o número total de bares-casas que teremos no problema dado. Dependendo do char de entrada, temos então a repassagem do flow do código para seu respectivo método.

2.1. Força-Bruta

O método da Força Bruta parte do pressuposto de enumerar todos os possíveis candidatos da solução e checar cada candidato para saber se ele satisfaz o enunciado do problema. Para o desenvolvimento do meu algoritmo de força bruta, tratei o problema com a criação de todas as possibilidades existentes que poderiam ser a solução do problema utilizando a criação de subgrupos, que seriam combinações diferentes do grupo total (que é o conjunto total de bares-casas). Para tal, não comparo o grupo de tamanho 01 elemento, pois este sempre será uma solução possível, uma vez que não realiza cruzamentos; a análise começa do subgrupo do tamanho seguinte, que é de 02 até n (n sendo o tamanho do conjunto total de bares-casas). Na combinação, a ordem na qual os elementos são dispostos não interfere no resultado, então temos uma representação-modelo abaixo para uma abstração do método seguindo um conjunto total de 04 elementos casa-bar {1,2,3,4}:

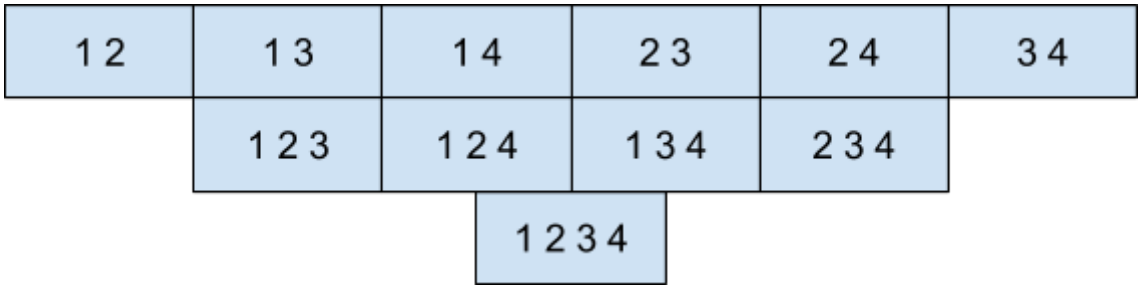


Figura 01: Subconjuntos de 02 até N elementos (Força Bruta)

A pirâmide invertida acima representa todas as possibilidades de subconjuntos possíveis de combinação do problema-exemplo em todos os tamanhos possíveis, deixando de lado os subconjuntos de tamanho 01, que são sempre soluções viáveis.

Para a implementação em alto nível, o *flow* do código começa recebendo, do *main.c*, o número total do conjunto de bares-casas, e segue para a função *forcaBruta*, que realiza a leitura das próximas n (no caso, número de bares-casas) linhas de entrada, as quais estão

explicitadas dois inteiros, que indicam o bar e a casa. Para minha resolução, separo em par e ímpar este conjunto de dois inteiros, uma vez que assim terei os dois lados da rua. Seguindo isso, inicializo então a criação dos subgrupos possíveis com todas as entradas dadas, começando do 02 (já explicado o porquê) até o tamanho total do número de elementos do grupo dado. Para a lógica dos subgrupos, chamo uma função chamada *recursaoCombinacao*, que para cada tamanho de subgrupo dado, gera, em um loop, todas as possíveis combinações e vai conferindo se o subconjunto criado é ou não uma solução. No caso de achar uma solução (ou seja, um subconjunto de itens que satisfaz a solução), o loop de busca por possíveis combinações é quebrado e tem-se então uma resposta válida para aquele tamanho de subgrupo. No caso de não achar solução, retorna-se 0, e no caso de achar, retorna-se o tamanho do subgrupo. Sempre que temos o retorno para cada tamanho de subgrupo, conferimos se ele é o maior número possível que pode ser obtido. Quando encerramos o loop da criação dos subgrupos, teremos, então, o maior subgrupo possível que responde ao problema dado, sendo ele a resposta do problema.

2.2. Método Guloso

O Método Guloso é a técnica de projeto de algoritmos que tenta resolver o problema fazendo a escolha localmente ótima em cada fase com a esperança de encontrar um ótimo global. Para tal, foi pensado na ordenação dos bares-casa baseado no número de colisões que eles têm: o elemento com o maior número de colisões será reordenado para a primeira posição no vetor de bares-casas, seguindo uma ordem decrescente. Após a ordenação pelo método *qsort* (disponível na biblioteca padrão), anda-se com o leitor do vetor para uma casa à direita, desconsiderando o elemento com o maior número de colisões, e o processo é repetido até termos somente elementos com um número 0 de colisões (ou seja, quando o maior elemento lido tiver 0 colisões). O número de elementos com 0 colisões neste ponto corresponde à resposta. Segue o desenho abaixo que abstrai o processo da lógica utilizada:

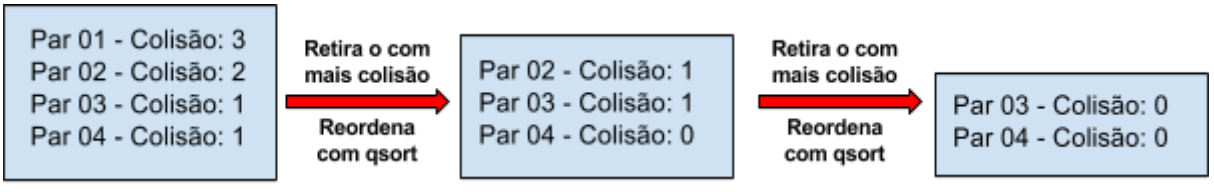


Figura 02: Lógica utilizada no Método Guloso

Para o flow do algoritmo, começamos recebendo, do *main.c*, o número total do conjunto de bares-casas, e segue para a função *guloso*, que realiza a leitura das próximas n

(no caso, número de bares-casas) linhas de entrada, as quais estão explicitadas dois inteiros, que indicam o bar e a casa. Para minha resolução, separo em par e ímpar este conjunto de dois inteiros, uma vez que assim terei os dois lados da rua. É então chamada a função *contaColisoes,* disponível no *auxiliares.c*, a qual, para cada elemento do vetor de bares-casas (aqui chamado de pares-impares, já que separo assim), é definido o número de colisões que ele tem no contexto dado (no caso, começamos com o número máximo de elementos disponíveis e vai retirando-se até o número-resposta). A lógica explicada (no primeiro parágrafo) é aplicada, até termos, por fim, o número total de elementos que se dispõem de maneira a nos apresentar um total de zero colisões - sendo esta a resposta.

Observe que o método guloso aplicado não é exato, e sim uma solução possível e aproximada para o problema, dando uma solução não-ótima com *razão de aproximação* de aproximadamente 0,8.

2.3. Programação Dinâmica

O método da Programação Dinâmica é um método para a construção de algoritmos para a resolução de problemas computacionais, em especial os de otimização combinatória. Ela é aplicável a problemas nos quais a solução ótima pode ser computada a partir da solução ótima previamente calculada e memorizada - de forma a evitar recálculo - de outros subproblemas que, sobrepostos, compõem o problema original. Para este método, utilizei a ideia do *Longest Increasing Subsequence* (ou LIS), que encontra a largura da maior subsequência de uma sequência dada tal que todos os elementos da subsequência são ordenados em ordem crescente. O método funciona, pois como um lado da rua já está ordenado, qualquer subsequência crescente do outro lado da rua garante que não haverá colisão, sendo assim a maior subsequência é a maior possibilidade de colocar bandeirolas sem que ocorra cruzamentos.

Para o flow do algoritmo, começamos recebendo, do *main.c*, o número total do conjunto de bares-casas, e segue para a função *dinamico*, que realiza a leitura das próximas n (no caso, número de bares-casas) linhas de entrada, as quais estão explicitadas dois inteiros, que indicam o bar e a casa. Após isso, escolho um lado para a ordenação (no caso, escolho o lado ímpar da “rua”), uma vez que preciso ordenar de maneira crescente, já que a rua é predisposta desta forma (na descrição do problema). Para o lado par, executo a *LIS*, que é a função que executa a lógica explicada (no primeiro parágrafo). Após isso, é printado o resultado final, que é o valor máximo do vetor auxiliar (criado na função LIS).

3. Análise de Complexidade

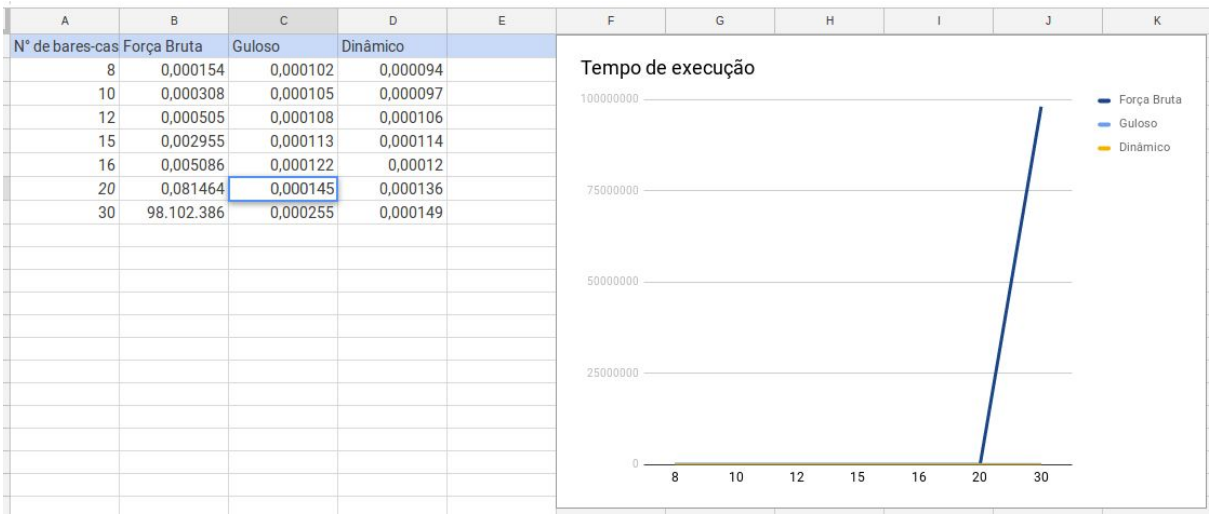
Nesta seção, será apresentada a análise do custo teórico de tempo e de espaço dos principais algoritmos e estruturas de dados utilizados. Portanto, foram analisadas as funções abaixo contidas na tabela:

Força Bruta	O algoritmo de Força Bruta faz uma análise combinatória sem repetição para criar todos os subgrupos possíveis para o número total de bares-casas, e a fórmula é um somatório de p começando em 2 até n de n!/(n-p)!, sendo n = número de elementos do conjunto total e p = tamanho do subgrupo. A complexidade fica em O(n!).
qsort	A complexidade do qsort, que é implementado pela biblioteca padrão do C, é O(n*log(n)).
Guloso	A complexidade do Guloso é baseada na sua maior estrutura custosa, que, no caso, é a contagem de colisões, ou seja, quando ocorre um cruzamento. Ela acontece no pior caso n vezes, com custo de n^2, assim sendo $O(n^3)$, sendo n = número de elementos do conjunto total.
Programação Dinâmica (LIS)	A complexidade do método dinâmico é baseada na sua maior estrutura custosa, que, no caso, é a contagem da LIS, ou seja, quando achar a subsequência crescente. Ela acontece no pior caso com custo de n^2, assim sendo $O(n^2)$, sendo n = número de elementos do conjunto total.

A **complexidade total do programa** depende do método usado - no melhor caso é o dinâmico, e no pior caso é o método da força bruta. A **complexidade espacial** depende do tamanho da entrada dada (ou seja, do número de pares bar-casa), ou seja, O(n).

4. Avaliações Experimentais

Os experimentos foram executados em um sistema com 8 GB de memória RAM e um processador Intel Core i7-3632QM CPU @ 2.20GHz. Para os testes, criei dois gráficos baseado no tempo de execução (segundos) versus o tamanho da entrada (número de pares bar-casa), a diferença sendo que o primeiro compara os três métodos e o segundo compara o Guloso e o Dinâmico apenas.



N° de bares-cas	Força Bruta	Guloso	Dinâmico
8	0,000154	0,000102	0,000094
10	0,000308	0,000105	0,000097
12	0,000505	0,000108	0,000106
15	0,002955	0,000113	0,000114
16	0,005086	0,000122	0,00012
20	0,081464	0,000145	0,000136
30	98.102.386	0,000255	0,000149

Figura 03: Guloso x Força Bruta x Dinâmico

No gráfico acima, é possível observar o pior desempenho do força bruta em comparação com os dois outros métodos, uma vez que ele explode seu valor de tempo de execução quando em testes pequenos.

N° de bares-cas	Guloso		Dinâmico
2600	53,782031		0,061807
3269	108,439676		0,088866

Figura 04: Guloso x Dinâmico

No gráfico acima, é possível observar o pior desempenho do método guloso em relação ao dinâmico em testes suficientemente grandes para observarmos uma diferença explícita.

5. Conclusão

O trabalho foi fundamental para a aplicabilidade de teorias que aprendi durante o curso - implementação de três abordagens de resolução de problema. A utilização do qsort também foi interessante para ampliar a ideia da base de bibliotecas disponíveis que podemos usar, bem como a criação de uma função própria para ser enviada para ele. O trabalho prático trouxe uma boa visualização de diferenças dos algoritmos, como também a medição prática do desempenho dos mesmos. Achei a proposta muito interessante e pude, com toda clareza, desenvolver minhas habilidades de lógica e programação.

6. Execução do Programa

Para executar corretamente o programa, é preciso utilizar o comando make, fazendo o uso do makefile para compilar, em seguida, a execução é dada por ./exec, e após isso é dado os dados de entrada.

7. Referências

http://www.geeksforgeeks.org/print-all-possible-combinations-of-r-elements-in-a-given-array-of-size-n/

https://www.youtube.com/watch?v=CE2b_-XfVDk

https://pt.wikipedia.org/wiki/Algoritmo_guloso

https://pt.wikipedia.org/wiki/Busca_por_for%C3%A7a_bruta

https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_din%C3%A2mica